

CS 162 Project 2: User Programs

March 27, 2015

Contents

1	Overview	3
2	Code style and commit messages	3
3	User Programs	3
4	Getting Started	4
5	Checkpoint 1	5
5.1	Background	5
5.1.1	Source Files	5
5.1.2	Using the File System	6
5.1.3	How User Programs Work	7
5.2	Requirements	7
5.2.1	Design Document	7
5.2.2	Design Review	7
5.2.3	Simple System Calls	7
5.2.4	Argument Passing	8
5.3	Tests	8
5.4	Scoring	8
6	Checkpoint 2	9
6.1	Background	9
6.1.1	Virtual Memory Layout	9
6.1.2	Accessing User Memory	10
6.2	Requirements	11
6.2.1	Accessing User Memory Correctly	11
6.2.2	System Calls: Processes	11
6.2.3	More Processes	12
6.2.4	Process Termination Messages	12
6.3	Tests	12
6.4	Scoring	13
7	Final Code and Report Handin	14
7.1	Background	14
7.2	Requirements	14
7.2.1	System Calls: File System	14
7.2.2	Denying Writes to Executables	15
7.3	Tests	15

7.4	Final Report	15
7.5	Scoring	15
8	ε Bonus	15
8.1	Other System Calls	15
9	References	16
9.1	80x86 Calling Convention	16
9.1.1	Program Startup Details	16
9.1.2	System Call Details	17
9.2	FAQ	18
9.2.1	Argument Passing FAQ	19
9.2.2	System Calls FAQ	19

1 Overview

In this project, you'll be implementing key parts of an operating system (Pintos) that allow user programs like wordcount to run. Along the way, you'll encounter familiar concepts, including:

- System Calls
- Processes/Threads
- File descriptors
- Userspace vs. Kernel
- Address Spaces

We divide this project to three checkpoints, each described separately with necessary background and checkpoint requirements. Every checkpoint is based on the understanding and implementation of former checkpoints.

In completing this project, we hope you'll have a more concrete understanding of these concepts and some hands-on experience with some actual code involved in running userspace programs.

2 Code style and commit messages

10 points of your final checkpoint 3 grade is "Efficient Design + Clean Code". Your GitHub commit messages and git hygiene are part of this criterion. **You should write your commit messages professionally, and you should never commit binaries/temporary files to GitHub, or you may lose a point here.**

3 User Programs

Now that you've worked with Pintos and are becoming familiar with its infrastructure and thread package, it's time to start working on the parts of the system that allow running user programs. The base code already supports loading and running user programs, but no I/O or interactivity is possible. In this project, you will enable programs to interact with the OS via system calls.

4 Getting Started

The skeleton we've added for project 2 is available in the group0 github repo, and you can work on project 2 on top of project 1 or start from scratch. If you're using Vagrant, you can pull the skeleton by running the following command:

```
cd ~/code/group
git checkout master
git pull staff master
cd src/userprog/
make
make check
```

If this doesn't work for you, double-check that the group0 repo is setup as a tracked repository:

```
git remote -v
```

You should see staff linked to the group0 repo. If not, add the group0 repo and try again:

```
git remote add staff git@github.com:Berkeley-CS162/group0.git
```

If you'd like to start from scratch, you can branch from staff/master:

```
git checkout -b <new-branch-name> staff/master
```

If these commands don't work for you or you aren't sure how to setup git, please try the following (most preferred solution first):

1. Read the spec
2. Read the spec again
3. Look on Google
4. Ask a classmate
5. Ask a GSI
6. Post a question on Piazza

5 Checkpoint 1

This checkpoint is due **Friday 3/20 11:59:59 PM**. Submit your code to the **release/proj2/checkpoint1** branch.

5.1 Background

User programs are written under the illusion that they have the entire machine, which means that the operating system must manage/protect machine resources correctly to maintain this illusion for multiple processes. In Pintos, more than one process can run at a time, but each process is single-threaded (multithreaded processes are not supported).

5.1.1 Source Files

In this project, you'll be working with a large number of files, primarily in the userprog directory. To help you parse through all the code, we've selected some key files and described them below:

process.c

process.h Loads ELF binaries and starts processes.

pagedir.c

pagedir.h A simple manager for 80x86 hardware page tables. Although you probably won't want to modify this code for this project, you may want to call some of its functions.

userprog/syscall.c

userprog/syscall.h Whenever a user process wants to access some kernel functionality, it invokes a system call. This is a skeleton system call handler. Currently, it terminates the user process in the case of exit and does nothing else for the other system calls.

lib/user/syscall.c

lib/user/syscall.h These functions provide a way for user processes to invoke each system call from a C program. Each uses a little inline assembly code to invoke the system call. We don't expect you to understand this assembly code, but we do expect you to understand the calling conventions described by your TA in section (also in Reference).

lib/syscall-nr.h This file defines the system call numbers for each system call.

exception.c

exception.h When a user process performs a privileged or prohibited operation, it traps into the kernel as an "exception" or "fault." (3) These files handle exceptions. Currently all exceptions simply print a message and terminate the process. Some, but not all, solutions to project 2 require modifying `page_fault()` in this file.

gdt.c

gdt.h The 80x86 is a segmented architecture. The Global Descriptor Table (GDT) is a table that describes the segments in use. These files set up the GDT. You should not need to modify these files for any of the projects. You can read the code if you're interested in how the GDT works.

tss.c

tss.h The Task-State Segment (TSS) is used for 80x86 architectural task switching. Pintos uses the TSS only for switching stacks when a user process enters an interrupt handler, as does Linux. You should not need to modify these files for any of the projects. You can read the code if you're interested in how the TSS works.

5.1.2 Using the File System

You will need to interface to the file system code for this project, because user programs are loaded from the file system and many of the system calls you must implement deal with the file system. However, the focus of this project is not the file system, so we have provided a simple but complete file system in the `filesys` directory. You will want to look over the `filesys.h` and `file.h` interfaces to understand how to use the file system, and especially its many limitations.

There is no need to modify the file system code for this project, and so we recommend that you do not. Working on the file system is likely to distract you from this project's focus.

Proper use of the file system routines will make life much easier, but we won't do it in this project. Because of that, you will have to tolerate the following limitations:

- No internal synchronization. Concurrent accesses will interfere with one another. You should use synchronization to ensure that only one process at a time is executing file system code.
- File size is fixed at creation time. The root directory is represented as a file, so the number of files that may be created is also limited.
- File data is allocated as a single extent, that is, data in a single file must occupy a contiguous range of sectors on disk. External fragmentation can therefore become a serious problem as a file system is used over time.
- No subdirectories.
- File names are limited to 14 characters.
- A system crash mid-operation may corrupt the disk in a way that cannot be repaired automatically. There is no file system repair tool anyway.

One important feature is included:

- Unix-like semantics for `filesys_remove()` are implemented. That is, if a file is open when it is removed, its blocks are not deallocated and it may still be accessed by any threads that have it open, until the last one closes it. See [Removing an Open File](#), for more information.

You need to be able to create a simulated disk with a file system partition. The `pintos-mkdisk` program provides this functionality. From the `userprog/build` directory, execute `pintos-mkdisk filesys.dsk --filesys-size=2`. This command creates a simulated disk named `filesys.dsk` that contains a 2 MB Pintos file system partition. Then format the file system partition by passing `-f -q` on the kernel's command line: `pintos -f -q`. The `-f` option causes the file system to be formatted, and `-q` causes Pintos to exit as soon as the format is done.

You'll need a way to copy files in and out of the simulated file system. The `pintos -p` ("put") and `-g` ("get") options do this. To copy file into the Pintos file system, use the command `pintos -p file -- -q`. (The `--` is needed because `-p` is for the `pintos` script, not for the simulated kernel.) To copy it to the Pintos file system under the name `newname`, add `-a newname`: `pintos -p file -a newname -- -q`. The commands for copying files out of a VM are similar, but substitute `-g` for `-p`.

Incidentally, these commands work by passing special commands `extract` and `append` on the kernel's command line and copying to and from a special simulated "scratch" partition. If you're very curious, you can look at the `pintos` script as well as `filesys/fsutil.c` to learn the implementation details.

Here's a summary of how to create a disk with a file system partition, format the file system, copy the `echo` program into the new disk, and then run `echo`, passing argument `x`. (Argument passing won't work until you implemented it.) It assumes that you've already built the examples in `examples` and that the current directory is `userprog/build`:

```
pintos-mkdisk filesys.dsk --filesys-size=2
pintos -f -q
pintos -p ../../examples/echo -a echo -- -q
pintos -q run 'echo x'
```

The three final steps can actually be combined into a single command:

```
pintos-mkdisk filesystems.dsk --filesystem-size=2
pintos -p ../../examples/echo -a echo -- -f -q run 'echo x'
```

If you don't want to keep the file system disk around for later use or inspection, you can even combine all four steps into a single command. The `--filesystem-size=n` option creates a temporary file system partition approximately `n` megabytes in size just for the duration of the `pintos` run. The Pintos automatic test suite makes extensive use of this syntax:

```
pintos --filesystem-size=2 -p ../../examples/echo -a echo -- -f -q run 'echo x'
```

You can delete a file from the Pintos file system using the `rm` file kernel action, e.g. `pintos -q rm file`. Also, `ls` lists the files in the file system and `cat file` prints a file's contents to the display.

5.1.3 How User Programs Work

Pintos can run normal C programs, as long as they fit into memory and use only the system calls you implement. Notably, `malloc()` cannot be implemented because none of the system calls required for this project allow for memory allocation. Pintos also can't run programs that use floating point operations, since the kernel doesn't save and restore the processor's floating-point unit when switching threads.

The `src/examples` directory contains a few sample user programs. The Makefile in this directory compiles the provided examples, and you can edit it to compile your own programs as well. Pintos can load *ELF* executables with the loader provided for you in `userprog/process.c`.

Until you copy a test program to the simulated file system (see 5.1.2 Using the File System), Pintos will be unable to do useful work. You should create a clean reference file system disk and copy that over whenever you trash your `filesystems.dsk` beyond a useful state, which may happen occasionally while debugging.

5.2 Requirements

5.2.1 Design Document

The design template for this project located on Piazza will contain specific questions designed to get you thinking and reading about the code in this project. It will also include space for you to detail how your group is planning to implement the functionality for the next two checkpoints. Please attach your design document in an email to your GSI in similar fashion to project 1.

5.2.2 Design Review

Setup a 30-minute meeting with your GSI. You should be able to answer questions about the project.

5.2.3 Simple System Calls

To support user programs, the operating system must be able to provide requested resources/functionality via syscalls, which are handled by the system call handler in `userprog/syscall.c`. The skeleton implementation provided "handles" system calls by terminating the process if the `exit` syscall was passed in. You will need to add functionality to the syscall handler to do the following:

1. Retrieve the system call number
2. Retrieve system call arguments
3. Carry out the appropriate actions described below for each syscall:

Note: Because there are a large number of system calls, it is much better style to use a table of system calls or switch-case statement rather than a huge nested if-then-else block (see the shell commands in hw1 for a good example of this).

Also, in preparation for future checkpoints, please implement the following system calls.

System Call: `int null (int i)` A "fake" system call that doesn't exist in any modern operating system.

You will implement this to get familiar with the system call interface. This system call increments the passed in integer argument by 1 and returns it to the user.

System Call: `int write (int fd, const void *buffer, unsigned size)` A dummy version of write that prints the userspace pointer passed in to stdout, trusting that the userspace pointer is safe to access. Returns the number of bytes actually written, which may be less than size if some bytes could not be written.

System Call: `void exit (int status)` Terminates the current user program, returning status to the kernel.

5.2.4 Argument Passing

Currently, `process_execute()` does not support passing arguments to new processes. Every user program will page fault immediately until argument passing is implemented. Implement argument passing by extending `process_execute()` so that instead of simply taking a program file name as its argument, it divides it into words at spaces. The first word is the program name, the second word is the first argument, and so on. That is, `process_execute("grep foo bar")` should run grep passing two arguments foo and bar.

Within a command line, multiple spaces are equivalent to a single space, so that `process_execute("grep foo bar")` is equivalent to our original example. You can impose a reasonable limit on the length of the command line arguments. For example, you could limit the arguments to those that will fit in a single page (4 kB). (Do not base your limit on the maximum 128 byte command-line arguments that the pintos utility can pass to the kernel.)

You can parse argument strings any way you like. If you're lost, look at `strtok_r()`, prototyped in `lib/string.h` and implemented with thorough comments in `lib/string.c`. You can find more about it by looking at the man page (run `man strtok_r` at the prompt).

See section 9.1.1 Program Startup Details, for information on exactly how you need to set up the stack.

After this checkpoint, your group should have an easy way to add future syscalls to your existing syscall handler (the cleanest approach is to use a table).

5.3 Tests

The following pintos tests should pass for Checkpoint 1: (try `./prepare ch1` in the `userprog/` directory)

```
exit
iloveos
null-test
args-dbl-space
args-many
args-multiple
args-none
args-single
```

5.4 Scoring

Design Document: 10 points

Design Review: 10 points

Simple System Calls: 5 points

6 Checkpoint 2

This checkpoint is due **Friday 4/3 11:59:59 PM**. Submit your code to the **release/proj2/checkpoint2** branch.

6.1 Background

6.1.1 Virtual Memory Layout

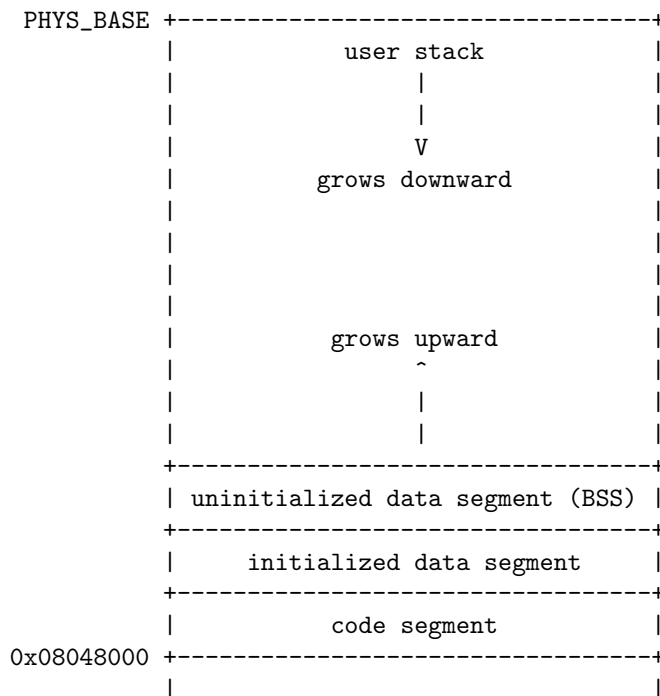
Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address 0 up to `PHYS_BASE`, which is defined in `threads/vaddr.h` and defaults to `0xC0000000` (3 GB). Kernel virtual memory occupies the rest of the virtual address space, from `PHYS_BASE` up to 4 GB.

User virtual memory is per-process. When the kernel switches from one process to another, it also switches user virtual address spaces by changing the processor's page directory base register (see `pagedir_activate()` in `userprog/pagedir.c`). `struct thread` contains a pointer to a process's page table.

Kernel virtual memory is global. It is always mapped the same way, regardless of what user process or kernel thread is running. In Pintos, kernel virtual memory is mapped one-to-one to physical memory, starting at `PHYS_BASE`. That is, virtual address `PHYS_BASE` accesses physical address 0, virtual address `PHYS_BASE + 0x1234` accesses physical address `0x1234`, and so on up to the size of the machine's physical memory.

A user program can only access its own user virtual memory. An attempt to access kernel virtual memory causes a page fault, handled by `page_fault()` in `userprog/exception.c`, and the process will be terminated. Kernel threads can access both kernel virtual memory and, if a user process is running, the user virtual memory of the running process. However, even in the kernel, an attempt to access memory at an unmapped user virtual address will cause a page fault.

Typical Memory Layout Conceptually, each process is free to lay out its own user virtual memory however it chooses. In practice, user virtual memory is laid out like this:





6.1.2 Accessing User Memory

As part of a system call, the kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above `PHYS_BASE`). All of these types of invalid pointers must be rejected without harm to the kernel or other running processes, by terminating the offending process and freeing its resources.

There are at least two reasonable ways to do this correctly. The first method is to verify the validity of a user-provided pointer, then dereference it. If you choose this route, you'll want to look at the functions in `userprog/pagedir.c` and in `threads/vaddr.h`. This is the simplest way to handle user memory access.

The second method is to check only that a user pointer points below `PHYS_BASE`, then dereference it. An invalid user pointer will cause a “page fault” that you can handle by modifying the code for `page_fault()` in `userprog/exception.c`. This technique is normally faster because it takes advantage of the processor's MMU, so it tends to be used in real kernels (including Linux).

In either case, you need to make sure not to “leak” resources. For example, suppose that your system call has acquired a lock or allocated memory with `malloc()`. If you encounter an invalid user pointer afterward, you must still be sure to release the lock or free the page of memory. If you choose to verify user pointers before dereferencing them, this should be straightforward. It's more difficult to handle if an invalid pointer causes a page fault, because there's no way to return an error code from a memory access. Therefore, for those who want to try the latter technique, we'll provide a little bit of helpful code:

```

/* Reads a byte at user virtual address UADDR.
   UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault
   occurred. */
static int
get_user (const uint8_t *uaddr)
{
    int result;
    asm ("movl $1f, %0; movzbl %1, %0; 1:"
        : "=a" (result) : "m" (*uaddr));
    return result;
}

/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
   Returns true if successful, false if a segfault occurred. */
static bool
put_user (uint8_t *udst, uint8_t byte)
{
    int error_code;
    asm ("movl $1f, %0; movb %b2, %1; 1:"
        : "=a" (error_code), "=m" (*udst) : "q" (byte));
    return error_code != -1;
}

```

6.2 Requirements

6.2.1 Accessing User Memory Correctly

Implement one of the two approaches described in section 6.1.2 Accessing User Memory.

6.2.2 System Calls: Processes

In this checkpoint, you will be writing the `wait`, `exec`, and `exit` system calls. In `pintos`, there is a 1 to 1 mapping between processes and threads. Thus, add structs to `thread.h` to encapsulate logic needed for communication between child and parent processes for waiting and exiting. Implement the following system calls.

System Call: `void halt (void)` Terminates `Pintos` by calling `shutdown_power_off()` (declared in `threads/init.h`). This should be seldom used, because you lose some information about possible deadlock situations, etc.

System Call: `void exit (int status)` Terminates the current user program, returning `status` to the kernel. If the process's parent waits for it (see below), this is the `status` that will be returned. Conventionally, a `status` of 0 indicates success and nonzero values indicate errors. Every user program that finishes in the normal way calls `exit` - even a program that returns from `main()` calls `exit` indirectly (see `start()` in `lib/user/entry.c`).

System Call: `pid_t exec (const char *cmd_line)` Runs the executable whose name is given in `cmd_line`, passing any given arguments, and returns the new process's program id (`pid`). Must return `pid -1`, which otherwise should not be a valid `pid`, if the program cannot load or run for any reason. Thus, the parent process cannot return from the `exec` until it knows whether the child process successfully loaded its executable. You must use appropriate synchronization to ensure this. For this checkpoint, your `exec` is allowed to work without the command-line, but you must implement the functionality necessary for `exec` to work with `wait`.

System Call: `int wait (pid_t pid)` Waits for a child process `pid` and retrieves the child's exit status.

If `pid` is still alive, waits until it terminates. Then, returns the `status` that `pid` passed to `exit`. If `pid` did not call `exit()`, but was terminated by the kernel (e.g. killed due to an exception), `wait(pid)` must return `-1`. It is perfectly legal for a parent process to wait for child processes that have already terminated by the time the parent calls `wait`, but the kernel must still allow the parent to retrieve its child's exit status, or learn that the child was terminated by the kernel.

`wait` must fail and return `-1` immediately if any of the following conditions is true:

- `pid` does not refer to a direct child of the calling process. `pid` is a direct child of the calling process if and only if the calling process received `pid` as a return value from a successful call to `exec`. Note that children are not inherited: if A spawns child B and B spawns child process C, then A cannot wait for C, even if B is dead. A call to `wait(C)` by process A must fail. Similarly, orphaned processes are not assigned to a new parent if their parent process exits before they do.
- The process that calls `wait` has already called `wait` on `pid`. That is, a process may wait for any given child at most once.

Processes may spawn any number of children, wait for them in any order, and may even exit without having waited for some or all of their children. Your design should consider all the ways in which waits can occur. All of a process's resources, including its struct `thread`, must be freed whether its parent ever waits for it or not, and regardless of whether the child exits before or after its parent.

You must ensure that Pintos does not terminate until the initial process exits. The supplied Pintos code tries to do this by calling `process_wait()` (in `userprog/process.c`) from `main()` (in `threads/init.c`). We suggest that you implement `process_wait()` according to the comment at the top of the function and then implement the wait system call in terms of `process_wait()`.

Warning: Implementing this system call requires considerably more work than any of the rest.

Note: You must synchronize system calls so that any number of user processes can make them at once. In particular, it is not safe to call into the file system code provided in the `filesys` directory from multiple threads at once. Your system call implementation must treat the file system code as a critical section. Don't forget that `process_execute()` also accesses files. For now, we recommend against modifying code in the `filesys` directory.

If a system call is passed an invalid argument, acceptable options include returning an error value (for those calls that return a value), returning an undefined value, or terminating the process.

6.2.3 More Processes

Implement the logic in `process_wait` and `process_exit` in `process.c`. How to do so was discussed in section during the week of March 19.

Next, implement the logic for `process_execute`. `process_execute` should create a new thread that calls `start_process`, which will try to load the binary program and if successful, initialize values for communication between child and parent processes. It will then start the user program.

To implement syscalls, you need to provide ways to read and write data in user virtual address space. You need this ability before you can even obtain the system call number, because the system call number is on the user's stack in the user's virtual address space. This can be a bit tricky: what if the user provides an invalid pointer, a pointer into kernel memory, or a block partially in one of those regions? You should handle these cases by terminating the user process. We recommend writing and testing this code before implementing any other system call functionality. See the 6.1.2 Accessing User Memory for more information.

When you're done with this part, Pintos should forevermore be bulletproof. Nothing that a user program can do should ever cause the OS to crash, panic, fail an assertion, or otherwise malfunction.

6.2.4 Process Termination Messages

Whenever a user process terminates, because it called `exit` or for any other reason, print the process's name and exit code, formatted as if printed by `printf ("%s: exit(%d)\n", ...)`. The name printed should be the full name passed to `process_execute()`, omitting command-line arguments. Do not print these messages when a kernel thread that is not a user process terminates, or when the `halt` system call is invoked. The message is optional when a process fails to load.

Aside from this, don't print any other messages that Pintos as provided doesn't already print. You may find extra messages useful during debugging, but they will confuse the grading scripts and thus lower your score.

6.3 Tests

The following pintos tests should produce correct output for Checkpoint 2. (try `./prepare ch2` in the `userprog/` directory)

```
wait-bad-pid
wait-killed
wait-simple
wait-twice
exec-multiple
```

exec-once
exec-missing
exec-bad-ptr
halt
exit

6.4 Scoring

Each test is worth 0.5 points for a combined score of 5 points.

7 Final Code and Report Handin

This checkpoint is due **Friday 4/10 11:59:59 PM**. Submit your code to the **release/proj2/checkpoint3** branch. The final report is due **Monday 4/13 11:59:59 PM**. Note that each slip-day for this checkpoint pushes both the final report and code deadlines back by one day. If you decide to use slip-days, please include the number of slip-days you are using in the final report submission (we will verify this number against your group's total as well as your code submission).

7.1 Background

There is no new background for this checkpoint.

7.2 Requirements

7.2.1 System Calls: File System

Implement the following system calls.

System Call: `bool create (const char *file, unsigned initial_size)` Creates a new file called `file` initially `initial_size` bytes in size. Returns true if successful, false otherwise. Creating a new file does not open it: opening the new file is a separate operation which would require a open system call.

System Call: `bool remove (const char *file)` Deletes the file called `file`. Returns true if successful, false otherwise. A file may be removed regardless of whether it is open or closed, and removing an open file does not close it. See Removing an Open File, for details.

System Call: `int open (const char *file)` Opens the file called `file`. Returns a nonnegative integer handle called a “file descriptor” (`fd`), or -1 if the file could not be opened.

File descriptors numbered 0 and 1 are reserved for the console: `fd 0 (STDIN_FILENO)` is standard input, `fd 1 (STDOUT_FILENO)` is standard output. The open system call will never return either of these file descriptors, which are valid as system call arguments only as explicitly described below.

Each process has an independent set of file descriptors. File descriptors are not inherited by child processes.

When a single file is opened more than once, whether by a single process or different processes, each open returns a new file descriptor. Different file descriptors for a single file are closed independently in separate calls to close and they do not share a file position.

System Call: `int filesize (int fd)` Returns the size, in bytes, of the file open as `fd`.

System Call: `int read (int fd, void *buffer, unsigned size)` Reads `size` bytes from the file open as `fd` into `buffer`. Returns the number of bytes actually read (0 at end of file), or -1 if the file could not be read (due to a condition other than end of file). `Fd 0` reads from the keyboard using `input_getc()`.

System Call: `int write (int fd, const void *buffer, unsigned size)` Writes `size` bytes from `buffer` to the open file `fd`. Returns the number of bytes actually written, which may be less than `size` if some bytes could not be written.

Writing past end-of-file would normally extend the file, but file growth is not implemented by the basic file system. The expected behavior is to write as many bytes as possible up to end-of-file and return the actual number written, or 0 if no bytes could be written at all.

`Fd 1` writes to the console. Your code to write to the console should write all of `buffer` in one call to `putbuf()`, at least as long as `size` is not bigger than a few hundred bytes. (It is reasonable to break up larger buffers.) Otherwise, lines of text output by different processes may end up interleaved on the console, confusing both human readers and our grading scripts.

System Call: void seek (int fd, unsigned position) Changes the next byte to be read or written in open file fd to position, expressed in bytes from the beginning of the file. (Thus, a position of 0 is the file's start.)

A seek past the current end of a file is not an error. A later read obtains 0 bytes, indicating end of file. A later write extends the file, filling any unwritten gap with zeros. (However, in Pintos files have a fixed length until project 4 is complete, so writes past end of file will return an error.) These semantics are implemented in the file system and do not require any special effort in system call implementation.

System Call: unsigned tell (int fd) Returns the position of the next byte to be read or written in open file fd, expressed in bytes from the beginning of the file.

System Call: void close (int fd) Closes file descriptor fd. Exiting or terminating a process implicitly closes all its open file descriptors, as if by calling this function for each one.

7.2.2 Denying Writes to Executables

Add code to deny writes to files in use as executables. Many OSes do this because of the unpredictable results if a process tried to run code that was in the midst of being changed on disk.

You can use `file_deny_write()` to prevent writes to an open file. Calling `file_allow_write()` on the file will re-enable them (unless the file is denied writes by another opener). Closing a file will also re-enable writes. Thus, to deny writes to a process's executable, you must keep it open as long as the process is still running.

7.3 Tests

All of the 78 Pintos userprog tests should pass for this checkpoint. We will weight each of these tests equally in determining your final score. (try `./prepare ch3` in the `userprog/` directory)

7.4 Final Report

For your final report, please email your GSI with a polished report describing your implementation of the project, following the template in the initial design. **Please create a new section in this document called "Final Design Changes" detailing the major changes to your design and why you changed them.**

7.5 Scoring

- 78 Pintos Tests, equally weighted - 50 points
- Final Report - 10 points
- Efficient Design + Clean Code - 10 points

8 ϵ Bonus

ϵ Bonus doesn't have any points.

8.1 Other System Calls

Implement other system calls in pintos codes.

9 References

This section contains a lot of information, some of which is required for understanding the project, and some of which is optional to read.

9.1 80x86 Calling Convention

This section summarizes important points of the convention used for normal function calls on 32-bit 80x86 implementations of Unix. Some details are omitted for brevity. If you do want all the details, refer to [SysV-i386].

The calling convention works like this:

1. The caller pushes each of the function's arguments on the stack one by one, normally using the PUSH assembly language instruction. Arguments are pushed in right-to-left order.
The stack grows downward: each push decrements the stack pointer, then stores into the location it now points to, like the C expression `*--sp = value`.
2. The caller pushes the address of its next instruction (the *return address*) on the stack and jumps to the first instruction of the callee. A single 80x86 instruction, CALL, does both.
3. The callee executes. When it takes control, the stack pointer points to the return address, the first argument is just above it, the second argument is just above the first argument, and so on.
4. If the callee has a return value, it stores it into register EAX.
5. The callee returns by popping the return address from the stack and jumping to the location it specifies, using the 80x86 RET instruction.
6. The caller pops the arguments off the stack.

Consider a function `f()` that takes three int arguments. This diagram shows a sample stack frame as seen by the callee at the beginning of step 3 above, supposing that `f()` is invoked as `f(1, 2, 3)`. The initial stack address is arbitrary:

```

                                +-----+
                                |          |
0xbffffe7c |          3          |
0xbffffe78 |          2          |
0xbffffe74 |          1          |
stack pointer --> 0xbffffe70 | return address |
                                +-----+
```

9.1.1 Program Startup Details

The Pintos C library for user programs designates `_start()`, in `lib/user/entry.c`, as the entry point for user programs. This function is a wrapper around `main()` that calls `exit()` if `main()` returns:

```
void
_start (int argc, char *argv[])
{
    exit (main (argc, argv));
}
```


The kernel must put the arguments for the initial function on the stack before it allows the user program to begin executing. The arguments are passed in the same way as the normal calling convention (see section 9.1 80x86 Calling Convention).

Consider how to handle arguments for the following example command: `/bin/ls -l foo bar`. First, break the command into words: `/bin/ls`, `-l`, `foo`, `bar`. Place the words at the top of the stack. Order doesn't matter, because they will be referenced through pointers.

Then, push the address of each string plus a null pointer sentinel, on the stack, in right-to-left order. These are the elements of `argv`. The null pointer sentinel ensures that `argv[argc]` is a null pointer, as required by the C standard. The order ensures that `argv[0]` is at the lowest virtual address. Word-aligned accesses are faster than unaligned accesses, so for best performance round the stack pointer down to a multiple of 4 before the first push.

Then, push `argv` (the address of `argv[0]`) and `argc`, in that order. Finally, push a fake "return address": although the entry function will never return, its stack frame must have the same structure as any other.

The table below shows the state of the stack and the relevant registers right before the beginning of the user program, assuming `PHYS_BASE` is `0xc0000000`:

Address	Name	Data	Type
0xbfffffff	<code>argv[3][...]</code>	<code>bar\0</code>	<code>char[4]</code>
0xbffffff8	<code>argv[2][...]</code>	<code>foo\0</code>	<code>char[4]</code>
0xbffffff5	<code>argv[1][...]</code>	<code>-l\0</code>	<code>char[3]</code>
0xbffffffd	<code>argv[0][...]</code>	<code>/bin/ls\0</code>	<code>char[8]</code>
0xbfffffec	<code>word-align</code>	0	<code>uint8_t</code>
0xbffffe8	<code>argv[4]</code>	0	<code>char *</code>
0xbffffe4	<code>argv[3]</code>	0xbffffffc	<code>char *</code>
0xbffffe0	<code>argv[2]</code>	0xbffffff8	<code>char *</code>
0xbffffdc	<code>argv[1]</code>	0xbffffff5	<code>char *</code>
0xbffffd8	<code>argv[0]</code>	0xbffffffd	<code>char *</code>
0xbffffd4	<code>argv</code>	0xbffffd8	<code>char **</code>
0xbffffd0	<code>argc</code>	4	<code>int</code>
0xbffffcc	<code>return address</code>	0	<code>void (*) ()</code>

In this example, the stack pointer would be initialized to `0xbffffcc`.

As shown above, your code should start the stack at the very top of the user virtual address space, in the page just below virtual address `PHYS_BASE` (defined in `threads/vaddr.h`).

You may find the non-standard `hex_dump()` function, declared in `<stdio.h>`, useful for debugging your argument passing code. Here's what it would show in the above example:

```

bffffffc0                                00 00 00 00 |          ....|
bffffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |.....|
bfffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 2f 62 69 |...../bi|
bffffff0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|

```

9.1.2 System Call Details

The first project already dealt with one way that the operating system can regain control from a user program: interrupts from timers and I/O devices. These are "external" interrupts, because they are caused by entities outside the CPU. The operating system also deals with software exceptions, which are events that occur in program code. These can be errors such as a page fault or division by zero. Exceptions are also the means by which a user program can request services ("system calls") from the operating system.

In the 80x86 architecture, the `int` instruction is the most commonly used means for invoking system calls. This instruction is handled in the same way as other software exceptions. In Pintos, user programs invoke

`int $0x30` to make a system call. The system call number and any additional arguments are expected to be pushed on the stack in the normal fashion before invoking the interrupt (see section 9.1 80x86 Calling Convention).

Thus, when the system call handler `syscall_handler()` gets control, the system call number is in the 32-bit word at the caller's stack pointer, the first argument is in the 32-bit word at the next higher address, and so on. The caller's stack pointer is accessible to `syscall_handler()` as the `esp` member of the `struct intr_frame` passed to it. (`struct intr_frame` is on the kernel stack.)

The 80x86 convention for function return values is to place them in the EAX register. System calls that return a value can do so by modifying the `eax` member of `struct intr_frame`.

You should try to avoid writing large amounts of repetitive code for implementing system calls. Each system call argument, whether an integer or a pointer, takes up 4 bytes on the stack. You should be able to take advantage of this to avoid writing much near-identical code for retrieving each system call's arguments from the stack.

9.2 FAQ

The following questions have been frequently asked by **Stanford** students in their operating systems class, and even though we know you're 1000 times more intelligent, we've included them here as reference.

How much code will I need to write? Here's a summary of our reference solution, produced by the `diffstat` program. The final row gives total lines inserted and deleted; a changed line counts as both an insertion and a deletion.

The reference solution represents just one possible solution. Many other solutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not modify all the files modified by the reference solution, and some may modify files not modified by the reference solution.

```
threads/thread.c      | 13
threads/thread.h      | 26 +
userprog/exception.c  | 8
userprog/process.c    | 247 ++++++-----
userprog/syscall.c    | 468 ++++++-----
userprog/syscall.h    | 1
6 files changed, 725 insertions(+), 38 deletions(-)
```

The kernel always panics when I run `pintos -p file -- -q`. Did you format the file system (with `pintos -f`)?

Is your file name too long? The file system limits file names to 14 characters. A command like `pintos -p ../../examples/echo -- -q` will exceed the limit. Use `pintos -p ../../examples/echo -a echo -- -q` to put the file under the name `echo` instead.

Is the file system full?

Does the file system already contain 16 files? The base Pintos file system has a 16-file limit.

The file system may be so fragmented that there's not enough contiguous space for your file.

When I run `pintos -p ./file --`, file isn't copied. Files are written under the name you refer to them, by default, so in this case the file copied in would be named `./file`. You probably want to run `pintos -p ./file -a file --` instead.

You can list the files in your file system with `pintos -q ls`.

All my user programs die with page faults. This will happen if you haven't implemented argument passing (or haven't done so correctly). The basic C library for user programs tries to read `argc` and `argv` off the stack. If the stack isn't properly set up, this causes a page fault.

All my user programs die with system call! You'll have to implement system calls before you see anything else. Every reasonable program tries to make at least one system call (`exit()`) and most programs make more than that. Notably, `printf()` invokes the write system call. The default system call handler just prints `system call!` and terminates the program. Until then, you can use `hex_dump()` to convince yourself that argument passing is implemented correctly (see section 9.1.1 Program Startup Details).

How can I disassemble user programs? The `objdump` (80x86) or `i386-elf-objdump` (SPARC) utility can disassemble entire user programs or object files. Invoke it as `objdump -d file`. You can use GDB's `disassemble` command to disassemble individual functions (see section E.5 GDB).

Why do many C include files not work in Pintos programs?

Can I use libfoo in my Pintos programs? The C library we provide is very limited. It does not include many of the features that are expected of a real operating system's C library. The C library must be built specifically for the operating system (and architecture), since it must make system calls for I/O and memory allocation. (Not all functions do, of course, but usually the library is compiled as a unit.)

The chances are good that the library you want uses parts of the C library that Pintos doesn't implement. It will probably take at least some porting effort to make it work under Pintos. Notably, the Pintos user program C library does not have a `malloc()` implementation.

How do I compile new user programs? Modify `src/examples/Makefile`, then run `make`.

Can I run user programs under a debugger? Yes, with some limitations. See section E.5 GDB.

What's the difference between `tid_t` and `pid_t`? A `tid_t` identifies a kernel thread, which may have a user process running in it (if created with `process_execute()`) or not (if created with `thread_create()`). It is a data type used only in the kernel.

A `pid_t` identifies a user process. It is used by user processes and the kernel in the `exec` and `wait` system calls.

You can choose whatever suitable types you like for `tid_t` and `pid_t`. By default, they're both `int`. You can make them a one-to-one mapping, so that the same values in both identify the same process, or you can use a more complex mapping. It's up to you.

9.2.1 Argument Passing FAQ

Isn't the top of stack in kernel virtual memory? The top of stack is at `PHYS_BASE`, typically `0xc0000000`, which is also where kernel virtual memory starts. But before the processor pushes data on the stack, it decrements the stack pointer. Thus, the first (4-byte) value pushed on the stack will be at address `0xbffffffc`.

Is `PHYS_BASE` fixed? No. You should be able to support `PHYS_BASE` values that are any multiple of `0x10000000` from `0x80000000` to `0xf0000000`, simply via recompilation.

9.2.2 System Calls FAQ

Can I just cast a `struct file *` to get a file descriptor?

Can I just cast a struct `thread *` to a `pid_t`? You will have to make these design decisions yourself. Most operating systems do distinguish between file descriptors (or pids) and the addresses of their kernel data structures. You might want to give some thought as to why they do so before committing yourself.

Can I set a maximum number of open files per process? It is better not to set an arbitrary limit. You may impose a limit of 128 open files per process, if necessary.

What happens when an open file is removed? You should implement the standard Unix semantics for files. That is, when a file is removed any process which has a file descriptor for that file may continue to use that descriptor. This means that they can read and write from the file. The file will not have a name, and no other processes will be able to open it, but it will continue to exist until all file descriptors referring to the file are closed or the machine shuts down.

How can I run user programs that need more than 4 kB stack space? You may modify the stack setup code to allocate more than one page of stack space for each process. This is not required in this project.

What should happen if an `exec` fails midway through loading? `exec` should return -1 if the child process fails to load for any reason. This includes the case where the load fails part of the way through the process (e.g. where it runs out of memory in the multi-oom test). Therefore, the parent process cannot return from the `exec` system call until it is established whether the load was successful or not. The child must communicate this information to its parent using appropriate synchronization, such as a semaphore, to ensure that the information is communicated without race conditions.